A LITRERATURE SURVEY ON VARIOUS SOFTWARE COMPLEXITY MEASURES

[1]*Anurag Bhatnagar,* [2]*Nikhar Tak ,*[3]*Shweta Shukla*

Abstract - Latin word “complexus”, which signifies "entwined", "twisted together". This may be interpreted in the following way - in order to have a complex we need two or more components, which are joined in such a way that it is difficult to separate them. Similarly, the Oxford Dictionary defines something as "complex" if it is "made of (usually several) closely connected parts". Software systems change over time, so it is very difficult to understand and measure the effect of these changes and it is a very complex problem in most modern software systems that consist of thousands of program modules. Thus one of the central problems in software engineering is its inherited complexity. This paper is a survey on different Code based complexity measures such as Halsted Complexity Measure and MAC Cabe’s Cyclomatic Complexity Measure and Cognitive based complexity measures such as KLCID Complexity Measure, Cognitive Functional Complexity Measure and Cognitive Information Complexity Measure.

Keyword – Software complexity, Code based complexity measures, cognitive based complexity measures.

1. INTRODUCTION TO COMPLEXITY

Complexity is defined by the execution time and storage required to perform the computation. When the interacting system is a programmer, complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task. Complexity measures focus on designs and actual code. They assume there is a direct correlation between design complexity and design errors, and code complexity and latent defects. By recognizing the properties of each that correlate to their complexity, we can identify those high risk applications that either should be revised or subjected to additional testing. Those software properties which correlate to how complex its size, interfaces among modules (usually

measured as *fan-in*, the number of modules invoking a given application, or *fan-out*, the number of modules invoked by a given application), and structure (the number of paths within a module). Complexity metrics help determine the number and type of tests needed to cover the design (interfaces or calls) or coded logic (branches and statements). There are several accepted methods for measuring complexity, most of which can be calculated by using automated tools. Basili defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task [1]. Software complexity measures are based on program code disregarding comment and stylistic attributes such as indentation and naming conventions. Measures typically depend on program size, control structure, or the nature of module interfaces.

Software complexity can be measured by Direct Measures which is also known as internal attributes and Indirect Measures which is also known as external attributes. Direct Measures are measured directly such as Cost, effort, LOC, speed, memory. Indirect Measures can not be measured directly. Example - Functionality, quality, complexity, efficiency, reliability, maintainability.

"Open Reengineering" provides criterion for metrics selection. Today "Open Systems" are so popular because commercial software applications is that the user is guaranteed a certain level of interoperability - the applications work together in a common framework, and applications can be ported across hardware platforms with minimal impact. The open reengineering consist of the abstract model that state that software systems must be as independent as possible from the source code formatting and programming language. The objective is to be able to set complexity standards and interpret the resultant numbers uniformly across projects and languages. Once we calculate the complexity then that value should be the same whether in Ada, FORTRAN, or in some other programming language. Some of the very basic but important complexity measures are the number of lines of code also known as LOC (Line of Code), does not meet the open reengineering criterion, as it's tremendously sensitive to programming language, coding style, and textual formatting of the source code. The cyclomatic complexity measure, is example of open engineering as it measures the amount of decision logic in a source code function. Cyclomatic complexity is entirely independent of text formatting and is this is independent of programming language

since the same fundamental decision structure are available and uniformly used in all procedural programming languages.

2. CODE BASED COMPLEXITY MEASURES

Code based complexity measures, as its name is indicating are based on the code of the program and not on the comments and stylistic attributes of it. Code based measures are typically depends upon the program sizes, program flow graphs, or module interfaces such as Halstead's software science metrics [3] and the most widely known measure of cyclomatic complexity developed by McCabe [6]. However, Halstead's software metrics purely calculates the number of operators and operands, but it does not consider the internal structures of modules, while McCabe's cyclomatic complexity does not consider I/O's of a system as it is based on the flow chart of the program. It uses flow chart of the program and on the basis of nodes and edges it provides complexity of the program.

2.1 HALSTEAD COMPLEXITY MEASURE

A set of software metrics was developed by Maurice Halstead in 1977 to find the complexity of the procedural programs, it is used to measure a software component's complexity directly from source code with prominence on computational complexity [3]. Halstead's measures uses distinct and total number of operators and operands means it absorbed the operators (that means operations in a programming language that may vary or move the operands) and operands (different variables, constants, and addresses on which operation is to be performed) in a software component in order to measure complexity. Halstead makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code. Thus the Halstead complexity measure uses code of the software in order to calculate the complexity so it is known as code based complexity measure. The goal of Halstead complexity measure was to identify measurable properties of software or code, and the relations between them. The main problem with this method or we can say blockage with this is that it does not distinguish the differences among the same operators and among the same operands in a program. Besides, it ignores the nested structure and fails to analyze the case statement

when code is not accessible. Some of the important parameters used in measures of Halstead are based on four scalar numbers of programs. All these parameters are show in Table 1.

Table – 1. Meta Measures in Halstead Complexity measure

Factor	Meta Measure	Description
n1	$operators	The number of distinct operators
n2	$operands	The number of distinct operands
N1	$operators	The total number of operators
N2	$operands	The total number of operands

Six measures can be derived from table – 1 , as shown in Table 2. Halstead's software metrics were developed in the context of assembler language programs, and get into too small details of measurement. The metrics identified the need for three types of measures for software: basic, derived, and estimated measures. But, some of the measures and constants are arguable, e.g. *T* = E/18, and *B = E2/3 / 3000,* and the physical meanings of the measures are not very clear.

Table – 2. Derived Measures of Halstead's Software Metrics

Measure	Symbol	Formula
Program Length	N	N = N1 + N2
Program vocabulary	n	n = n1 + n2
Volume	V	$V = N*(Log_2 n)$
Difficulty	D	D = (n1*N2)/2*n2
Effort	E	E = D*V
Time	T	T = E/18
Total Bugs	B	$B = E^{2/3}/3000$

2.2 MAC CABE'S CYCLOMETRIC COMPLEXITY

In 1976, Thomas J.McCabe provides a method calculates Cyclomatic Complexity by the control flow graph of program. This McCabe method is based on control flow. McCabe's complexity measure is a mathematical technique for calculating the logical complexity of a computer program. **Cyclomatic complexity** also known as **conditional complexity,** is a software metric and is used to indicate the complexity of a program. It is a direct measure so it directly measures the number of linearly independent paths through a program's source code. Cyclomatic complexity is computed

using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. A number representing its logical weight can be used to show the complexity of a computer program. This quantitative complexity number dependent on a program's decision structure or the number of basic paths through a program generated but it is independent of the program's size. Complexity can assume a value of one to infinity, a reasonable upper limit of intellectual manageability has been placed by McCabe at ten. Software should be tested for the complexity once they are created but when the complexity of the software exceeds than ten, sub-functions should be given their own procedure or the software should be redone. The theoretical basis for McCabe's complexity measure is graph theory. The following connection exists between graph theory and computer programs. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. Thus, all computer programs may be expressed as graphs or to be precise "program control graphs." Thus mathematical analysis may be applied to computer programs in order to calculate complexity. Graph theory allows for such a graph to yield a quantitative cyc]omatic complexity number via the formula –

$$V(F) = E - N + 2$$

F is the flow graph of the code.
N is the number of vertices/Nodes.
E is the number of edges.

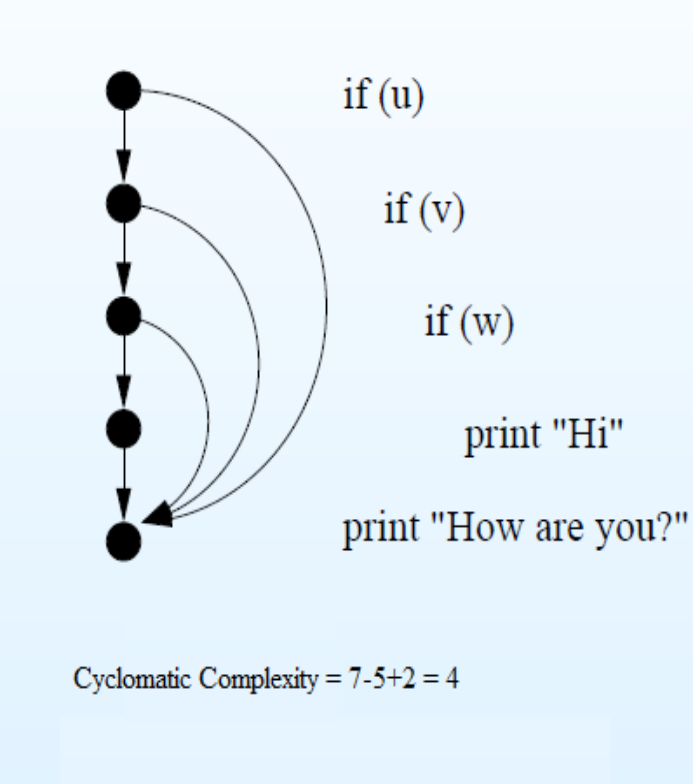

Figure – 1. McCabe Cyclomatic Complexity Example.

Here we have
E = No. of Edges = 7
N = No. of Nodes = 5.

Cyclomatic Complexity
V(F) = 7-5+2 = 4

3. COGNITIVE COMPLEXITY MEASURES

Cognitive complexity measures are the human effort needed to perform a task or difficulty in understanding the software. Cognitive complexity measure is an attempt that quantifies the effort or notch of difficulty in understanding the software based on cognitive informatics foundation. Cognitive complexity of software is dependent on three basic fundamental factors. These are inputs, outputs, and internal processing. Cognitive complexity measures consider all these factors affecting the effort in twigging the software, e.g. data objects such as inputs, outputs, variables, loops and branches evaluating complexity but if the factors are not carefully thought and organized it may cause trouble in calculating complexity. Some of the factors considered in order to calculate the complexity of the software and these factors are separately-derived weights. Thus this process of calculating the complexity ignores the relationships among the factors of the modules and has a little importance to human cognitive process when apprehending the code of the program.

3.1 KLCID COMPLEXITY METRICS

Klemola and Rilling proposed KLCID based complexity measure in 2004. It defines the use of the identifiers as programmer defined variables and identifiers (ID) when a software is built up.

ID = Total no. of identifiers/ LOC

In order to calculate KLCID, we need to find the number of unique lines of code in a module, lines that have same type and kind of operands with same arrangements of operators would be consider equal. I defines KLCID as –

KLCID= No. of Identifier in the set of unique lines/ No. of unique lines containing identifier

This is a time consuming method when comparing a line of code with each line of the program. KLCID accepts that internal control structures for different software's are identical.

3.2 COGINITIVE FUNCTIONAL SIZE (CFS)

Cognitive Functional Size (CFS) [1] was defined by Wang
As –

$$CFS = (N_i + N_o) * Wc$$

Where N_i = No of inputs.
N_o = No of outputs.
Wc = The total cognitive weight of basic control structures (BCS's).

These BCS are defined as the total sum of cognitive weights of its Q linear blocks composed in individual BCS's. Since each block may consist of 'm' layers of nesting BCS's, and each layer with 'n' linear BCS's, then –

$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right]$$

Table 2. Cognitive Weights (Wc) of BCS's

Category	BCS	Wc.
Sequence	Sequence (SEQ)	1
Branch	If-Then-Else (ITE)	2
	Case(CASE)	3
Iteration	For-do (Ri)	3
	Repeat-Until (Ri)	3
	While-do (Ro)	3
Embedded Component	Function call (FC)	2
Concurrency	Recursion (REC)	3
	Parallel (PAR)	4
	Interrupt (INT)	4

Cognitive Functional Size interestingly started the study in software complexity measurement based on cognitive informatics foundation [6] and state that the complexity of software is dependent on inputs, outputs, and its internal processing.

3.2 COGNITIVE INFORMATION COMPLEXITY MEASURE

Cognitive Informatics plays an important role in understanding the fundamental characteristics of software. Wang [8] defined information as the third essence in modelling the natural world supplement to matter and energy. Wang [9] also defined software as "Software in cognitive informatics is perceived as formally described design information and implementations instructions of computing application" i.e.

Software ≃ Information

It represent that, software is equivalent to information, So
implies that

Difficulty in understanding ≃ Difficulty in software understanding information

Software is a computational information and is a mathematical entity. Or we can say that software is a piece of information that consist of identifiers to represent the information and Operators that represent function to be performed.

Software = fun(Identifiers, Operators)

Identifiers can be variable names, defined constants or other labels in software. Thus information can be defined as –

Definition 1: Information can be represented by LOC. It means different operators and operand can be used to show information, Thus in K^{th} line of code the Information contained is –

$$\mathbf{I_{Kth} = (Identifiers + Operands)_K}$$
$$\mathbf{= (ID_K + OP_K)\ IU}$$

Where ID = Total number of identifiers in the k^{th} LOC of software.
OP = Total number of operators in the k^{th} LOC of software.
IU = Information Unit represents that any identifier or operator has one unit of information in it.

Definition 2: Overall **Information** contained in software (I_{total}) is the sum of information contained in each line of Code (LOCS) i.e.

$$\mathbf{I_{total} = \sum_{K=1}^{LOCS} I_k}$$

When have established that software is ready to comprehend the information units (IU's), the measure of the complexity of the software should contain the above parameters. Based on this fact, weighted information count is been introduced as in definition 3.

Definition 3: The **Weighted Information Count** of a **line of code (WICL)** of a software is a function of identifiers, operands and LOC and is defined as –

$$\mathbf{WICL_k = I_k / [LOCS – k]}$$

Where WIC_k = Weighted Information Count for the kth line.
I_k = Information contained in a software for the kth line. So the **Weighted Information Count of the Software (WICS)** is defined as (LOCS) –

$$\mathbf{WICS = \sum_{K=1}^{LOCS} WICL_k}$$

The internal control structure of software such as it's weight must be considered in order to provide complete and robust complexity measure.

Definition 4: Sum of the cognitive weights of basic control structures (SBCS) is defined –
Let W1, W2…..Wn be the cognitive weights of the basic control structures [8] in the software –

$$\mathbf{SBCS = \sum_{i=1}^{n} (W_i)}$$

Definition 5: Thus we have **Cognitive Information Complexity Measure –**

(CICM) = WICS * SBCS

Thus we have used an approach to measure the amount of information contained in the software thus enabling us to calculate the coding efficiency (EI**)**.

Definition 6: Efficiency of Information Coding (EI) of a software is defined as –

EI = I_{total} / LOCS.

Now to analyse the KLCID, CMS and CICM complexity measures, we have an algorithm to calculate the average of a set of 'n' numbers on which we will applied all these methods to calculate the complexity.

```
# define n 10
Void main( )
{
int total;
float sum, avg, num;
sum = 0;
total = 0;
while (total < n)
{
scanf("%f", &num);
sum = sum + num;
total = total + 1;
}
avg = sum/N;
printf("N=%d sum = %f"', N, sum);
printf("average = %f"', avg);
}
```

Calculation of KLCID complexity measure –

Total no. of identifiers in the above program = 18
Total no. of lines of code = 17
ID = 18/17 = 1.05
No. of unique lines containing identifier = 9
No. of identifiers in the set of unique lines = 11
KLCID = 11 / 9 = 1.22

Calculation of CFS –

Number of inputs Ni = 1
Number of outputs No = 3
BCS (sequence) W1 = 1
BCS (while) W2 = 3
Wc = W1 + W2 = 1+3 = 4
CFS = (Ni + No)* Wc = (1+3)*4 = 16

Calculation of CICM –

LOC = 17
Total no. of identifiers = 18
Total no. of operators = 4
BCS(sequence) W1 = 1
BCS(while) W2 = 3
SBCS = W1 + W2 = 1+3 = 4
WICS = [1/16 + 1/13 + 3/12 + 1/11 + 1/10 + 3/9 + 1/7 + 4/6 + 3/5 + 4/3] = 3.63
CICM = WICS * SBCS = 3.63 * 4 = 14.53
Information Coding Efficiency (EI) of the above
program = 22/17 = 1.29
The cognitive information complexity of the given algorithm is 14.53 CICU (Cognitive Information Complexity unit).

3.3 CONCLUSION

In this paper we have studied various Software complexity measures along with their appropriate example. All Code based complexity measures such as Halsted Complexity Measure and MAC Cabe's Cyclomatic Complexity Measure that we have studied are based on coding part of the program, so it needed time to compute complexity as all these methods can be applied after the coding phase has been completed. MAC Cab Cylomatic complexity measure is based on the flow graph of the program by which code is generated thus it is also classified in code based complexity measures. Cognitive based complexity measures such as KLCID Complexity Measure, Cognitive Functional Complexity Measure and Cognitive Information Complexity Measure are also based on the coding part of the programs so they also needed code of the procedural program on which they are to be applied, in order to compute the complexity. So we can say that the time to measure the complexity for procedural programs by both Code based complexity measures and Cognitive based complexity measures depends on the time that is needed to complete the coding phase.

3.4 REFERENCES

[1] B. Auprasert and Y, Limpiyakorn, "StructuringCognitive Information for Software Complexity Measurement", Accepted for CSIE 2009, Los Angeles, USA., April 2009.

[2] Halstead, M.H., Elements of Software Science, Elsevier North, New York, 1977

[3] Mc Cabe, T.H., A Complexity measure, IEEE Transactions on Software Engineering, SE-2,6, pp. 308-320, 1976

[4] Kushwaha, D.S. and Misra, A.K., A Modified Cognitive Information Complexity Measure of Software, ACM SIGSOFT Software Engineering Notes, Vol. 31, No. 1 January 2006.

[5] Kushwaha, D.S. and Misra, A.K., A Complexity Measure based on Information Contained in Software, Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006, (pp 187-195)

[6] Tumous Klemola and Juergen Rilling, A Cognitive Complexity Metric based on Category Learning, IEEE International Conference on Cognitive Informatics (ICCI-04)

[5] Kushwaha, D.S. and Misra, A.K., Cognitive Information Complexity Measure: A Metric Based on Information Contained in the Software, WSEAS Transactions on Computers, Issue 3, Vol. 5, March 2006, ISSN: 1109 – 2750

[6] Kushwaha, D.S. and Misra, A.K., Improved Cognitive Information Complexity Measure: A metric that establishes programcomprehension effort, ACM SIGSOFT Software Engineering, Page 1, September 2006, Volume 31 Number 5

[7]IEEE Computer Society: IEEE Recommended Practice for Software Requirement Specifications, New York, 1994

[8] Wang. Y and Shao,J.,"On Cognitive informatics", *1st IEEE International Conference on Cognitive Informatics*, pages 34-42,August 2002.

[9] Wang ,Y .and Shao,J., "Measurement of the Cognitive Functional Complexity of Software", *3 rd IEEE International Conference on Cognitive Informatics(ICCI'04).*